MAT 115

Exercise #6

Chapter 3 of the text covers programming basics. We will focus on conditional expressions (3.1), functions (3.2), and for-loops (3.4).

Conditional expressions use if-else statements. Let's look at the first example from the text. The goal of this example is to print the reciprocal of `a` unless `a` is 0:

```
a <- 2

if(a!=0){

  print(1/a)

  } else{

  print("No reciprocal for 0.")

  }
```

```
## [1] 0.5
```

What happens if we set `a` equal to 2?

A: It prints out .5 which is 1/2

Now lets look at an example using the `temp_carbon` dataset in the `dslabs` package. First, we need to load the package.

```
library(dslabs)

# Write the code to look at the help documentation for the temp_carbon data here.
help('temp_carbon')
```

```
## starting httpd help server ... done
```

The output from the example code below will indicate whether maximum land or ocean temperature anomalies are greater. Anomaly in this context is difference from 20th century mean values.

Before 1880, there is no anomaly data, so we must remove those rows before running the loop. The missing data here are identified with NAs.

```
temp_carbon_complete <- temp_carbon[which(temp_carbon$temp_anomaly!="NA"),]

if (max(temp_carbon_complete$land_anomaly) > max(temp_carbon_complete$ocean_anomaly)) {

  print("Max land temp anomaly is higher than max ocean temp anomaly")

} else {

    print("Max ocean temp anomaly is higher than max land temp anomaly")

  }
```

```
## [1] "Max land temp anomaly is higher than max ocean temp anomaly"
```

A related function called 'ifelse' is very useful because it works on vectors. The following example is similar to the first example from this exercise, but `a` is a vector.

```
a <- c(0, 1, 2, -4, 5)
ifelse(a > 0, 1/a, NA)
```

```
## [1]  NA 1.0 0.5  NA 0.2
```

What happens if you try to run the first example but with `a` as a vector instead of a scalar?

```
#b <- c(0, 1, 2, -4, 5)

#if(b!=0){

  #print(1/a)

  #} else{

  #print("No reciprocal for 0.")

  #}
```

A: It gives me an error because it is trying to compare a vector an numeric value.

Write an `ifelse` statement that returns "value > 0" for ocean anomaly values above 0 and "value <= 0" for values below 0.

```
# Write your code chunk here.
ifelse(temp_carbon_complete$ocean_anomaly > 0, print("value > 0"), print("value <= 0"))
```

```
## [1] "value > 0"
## [1] "value <= 0"
```

```
##   [1] "value <= 0" "value > 0"  "value <= 0" "value <= 0" "value <= 0"
##   [6] "value <= 0" "value <= 0" "value <= 0" "value <= 0" "value <= 0"
##  [11] "value <= 0" "value <= 0" "value <= 0" "value <= 0" "value <= 0"
##  [16] "value <= 0" "value <= 0" "value <= 0" "value <= 0" "value <= 0"
##  [21] "value <= 0" "value <= 0" "value <= 0" "value <= 0" "value <= 0"
##  [26] "value <= 0" "value <= 0" "value <= 0" "value <= 0" "value <= 0"
##  [31] "value <= 0" "value <= 0" "value <= 0" "value <= 0" "value <= 0"
##  [36] "value <= 0" "value <= 0" "value <= 0" "value <= 0" "value <= 0"
##  [41] "value <= 0" "value <= 0" "value <= 0" "value <= 0" "value <= 0"
##  [46] "value <= 0" "value <= 0" "value <= 0" "value <= 0" "value <= 0"
##  [51] "value <= 0" "value <= 0" "value <= 0" "value <= 0" "value <= 0"
##  [56] "value <= 0" "value <= 0" "value <= 0" "value <= 0" "value <= 0"
##  [61] "value > 0"  "value > 0"  "value > 0"  "value > 0"  "value > 0"
##  [66] "value > 0"  "value <= 0" "value <= 0" "value <= 0" "value <= 0"
##  [71] "value <= 0" "value > 0"  "value > 0"  "value > 0"  "value <= 0"
##  [76] "value <= 0" "value <= 0" "value > 0"  "value > 0"  "value > 0"
##  [81] "value > 0"  "value > 0"  "value > 0"  "value > 0"  "value <= 0"
```

```
##  [86] "value <= 0" "value > 0"  "value <= 0" "value > 0"  "value > 0"
##  [91] "value > 0"  "value <= 0" "value > 0"  "value > 0"  "value <= 0"
##  [96] "value <= 0" "value <= 0" "value > 0"  "value > 0"  "value > 0"
## [101] "value > 0"  "value > 0"  "value > 0"  "value > 0"  "value > 0"
## [106] "value > 0"  "value > 0"  "value > 0"  "value > 0"  "value > 0"
## [111] "value > 0"  "value > 0"  "value > 0"  "value > 0"  "value > 0"
## [116] "value > 0"  "value > 0"  "value > 0"  "value > 0"  "value > 0"
## [121] "value > 0"  "value > 0"  "value > 0"  "value > 0"  "value > 0"
## [126] "value > 0"  "value > 0"  "value > 0"  "value > 0"  "value > 0"
## [131] "value > 0"  "value > 0"  "value > 0"  "value > 0"  "value > 0"
## [136] "value > 0"  "value > 0"  "value > 0"  "value > 0"
```

Section 3.2 covers functions. One of the many reasons R is so powerful is that it has common operations (eg, `max`, `min`, `length`, `sum`) built into it as functions. However, you can also write your own functions, pretty cool!

The text does a good job of going through the form of a function. Here is an example that calculates mean:

```
avg <- function(x){

  s <- sum(x)

  n <- length(x)

  s/n

  }
```

Once you create the function, you can use it:

```
avg(a)
```

```
## [1] 0.8
```

```
avg(temp_carbon_complete$temp_anomaly)
```

```
## [1] 0.06
```

(Note that variables assigned inside the function are *NOT* saved in the workspace.)

Here is another example of a function that calculates the percent difference between two means:

```
perc_diff <- function(x,y) {

  xm <- mean(x)
  ym <- mean(y)

  z <- (xm-ym)/xm

  z

}
```

You can use it to compare the means of land and ocean anomalies across all the years:

```
perc_diff(temp_carbon_complete$land_anomaly,temp_carbon_complete$ocean_anomaly)
```

```
## [1] 0.2558376
```

Try modifying the `perc_diff` function so that it calculates the percent difference relative to the `y` variable instead of the `x` variable.

```
# Write your code chunk here.
perc_diff <- function(x,y) {

  xm <- mean(x)
  ym <- mean(y)

  z <- (ym-xm)/ym

  z

}

perc_diff(temp_carbon_complete$land_anomaly,temp_carbon_complete$ocean_anomaly)
```

```
## [1] -0.3437926
```

Section 3.4 covers for-loops. Imagine we want to calculate the median value for all columns in the `temp_carbon_complete` dataset. We could just use the `median` function once for each variable:

```
median(temp_carbon_complete$year)
```

```
## [1] 1949
```

```
median(temp_carbon_complete$temp_anomaly)
```

```
## [1] -0.03
```

```
median(temp_carbon_complete$land_anomaly)
```

```
## [1] -0.05
```

```
median(temp_carbon_complete$ocean_anomaly)
```

```
## [1] -0.01
```

```
median(temp_carbon_complete$ocean_carbon_emissions)
```

```
## NULL
```

Or we could use a for-loop.

```r
m <- vector(length=ncol(temp_carbon_complete)) # 1. output

for (i in 1:length(m)) {                       # 2. sequence

  m[i] <- median(temp_carbon_complete[,i])     # 3. body

}

m
```

```
## [1] 1949.00   -0.03   -0.05   -0.01      NA
```

(Note: we have some missing values in the carbon emission variable, so the median is returned as NA)

To help us understand what happened here, we can separate the loop into three parts:

1. The output: before you start the loop, you must always allocate sufficient space for the output. In this case the number of columns in the data frame. This is very important for efficiency: if you grow the for loop at each iteration using `c()` (for example), your for loop will be very slow.

A general way of creating an empty vector of given length is the `vector()` function. It has two arguments: the type of the vector ("logical", "integer", "double", "character", etc) and the length of the vector.

2. The sequence: this determines what to loop over: each run of the for loop will assign `i` to a different value from `1:length(m)`.
3. The body: this is the code that does the work. It's run repeatedly, each time with a different value for `i`. The first iteration calculates the median of the first column and so on.

If you are still having trouble wraping your head around the for-loop, here is an even simpler example:

```r
m <- 1:5                # output

for(i in 1:length(m)){ # sequence

  print(m[i])           # body

}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

Can you explain in your own words what this loop does?

A: The for loop starts at 1 and goes to 5 using i as a tracker in order to check if 5 has been reached or not. i is also used to access each index of m.

Try writing a for-loop that creates a vector of the difference between ocean and land anomalies for each year in the `temp_carbon_complete` dataset.

```
# Write your code chunk here.
#View(temp_carbon_complete)

m <- vector(length=nrow(temp_carbon_complete))

for(i in 1:length(m)){

  diff <- temp_carbon_complete$ocean_anomaly[i] - temp_carbon_complete$land_anomaly[i]
  m[i] <- diff
}

m
```

```
##   [1]  0.47  0.41  0.48  0.62  0.55  0.39  0.34  0.24  0.36  0.29  0.22  0.37
##  [13]  0.26  0.30  0.11  0.22  0.33  0.23  0.15  0.08  0.10 -0.04  0.02  0.00
##  [25] -0.02  0.08 -0.05  0.33  0.01 -0.02 -0.06  0.05  0.20 -0.01 -0.11 -0.02
##  [37]  0.20  0.34  0.29  0.12  0.18 -0.02  0.03  0.05  0.01 -0.04 -0.08  0.06
##  [49] -0.05  0.22  0.02 -0.06 -0.14  0.14 -0.07  0.02  0.00  0.01 -0.27 -0.13
##  [61]  0.13  0.25  0.07  0.05  0.13  0.37  0.00 -0.11 -0.15 -0.01  0.23  0.08
##  [73]  0.13 -0.10  0.03 -0.02  0.30  0.15 -0.03 -0.01  0.07 -0.03 -0.07 -0.13
##  [85]  0.11  0.07  0.06 -0.02  0.12  0.25  0.01 -0.07  0.28 -0.20  0.16 -0.18
##  [97]  0.22 -0.06  0.02  0.07 -0.05 -0.27  0.11 -0.20  0.14  0.08 -0.09 -0.09
## [109] -0.26 -0.09 -0.29 -0.19 -0.01 -0.10 -0.19 -0.43 -0.04 -0.18 -0.46 -0.47
## [121] -0.28 -0.38 -0.46 -0.42 -0.32 -0.58 -0.47 -0.69 -0.48 -0.36 -0.58 -0.47
## [133] -0.44 -0.50 -0.38 -0.64 -0.71 -0.66 -0.50
```

For-loops are a foundational programming tool. However, we rarely use them in R. Instead, we use vectorized functions because they typically result in shorter, clearer, and faster running code.

For example, instead of using a for-loop to calculate the medians of each column in the `temp_carbon_complete` dataset we can use the `apply` function:

```
apply(temp_carbon_complete,MARGIN = 2,FUN = median)
```

```
##             year     temp_anomaly     land_anomaly    ocean_anomaly
##          1949.00            -0.03            -0.05            -0.01
## carbon_emissions
##               NA
```

The `margin` argument applies the function to either rows (1) or columns (2) of the array (`temp_carbon_complete`).

Loops with complex computations or long sequences can take a long time to execute. Let's compare how long it takes for R to calculate the square root of all whole numbers between 1 and 1e+8 using a `for-loop` or just the `sqrt` function.

We can measure execution time with the `system.time` function. The execution time of code relates most closely to the *user* value given by 'system.time'. The vectorized function should be significantly faster.

```
system.time (for (i in 1:1e+9) {

sqrt(i)

})
```

```
##    user  system elapsed
##   51.94    0.10   55.16
```

```
rm(i)
```

```
system.time(sqrt(1:1e+9))
```

```
##    user  system elapsed
##   13.87    8.06   26.02
```

```
#system.time(sqrt(1:1e+11))
```

Try increasing and decreasing the order of magnitude of the numbers used in the calculations above.

How do the execution times change? Did any of them surprise you and why?

A: Anything below e+8 is almost a second less for just sqrt() and anything above e+8 takes significantly longer by about 8 more second. Also e+10 and above results in an error for me that says Error: cannot allocate vector of size [74.5] gb. (The brackets are there for larger e+ values)

Anything below e+8 for the for loop is also much faster but it is still not as fast as just using the sqrt() function. e+9 for the for loop took signicantly longer than the sqrt() function. It surprised me that it was almost a 41 second difference which is a gigantic gap.